САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ УПРАВЛЕНИЯ

Мишенин А.Н., Стученков А.Б.

КЛАССИФИКАЦИЯ ТЕКСТОВ

Часть 2:

Matplotlib. Pandas. Регулярные выражения. Обработка текстов

Учебно-методическое пособие

Санкт-Петербург
2020

УДК 025.4.03 : 004.738.5
ББК 32.971.353 : 32.973

А.Н. Мишенин, А.Б. Стученков

Классификация текстов. Часть 2: Matplotlib. Pandas. Регулярные выражения. Обработка текстов.

Учебно-методическое пособие – СПб., 2020. – 45с.

Учебно-методическое пособие разработано на основе опыта чтения курсов лекций и проведения практических занятий по учебным дисциплинам «Классификация документов» и «Информационный поиск» на факультете Прикладной Математики – Процессов Управления Санкт-Петербургского государственного университета. В второй части внимание уделено изучению базовых библиотек на научных вычислений на Python, а также регулярным выражениям и методам работы с текстом.

Содержание

Matplotlib, SciPy, SymPy

Matplotlib

Matplotlib[3] де факто стандарт для построения диаграмм и графиков в Python. В последнее время вытесняется более современными инструментами - plotly[7], d3[**???**] и т.д.

В данном примере отображается график синуса и косинуса

```
%matplotlib inline

import matplotlib.pyplot as plt
import seaborn as sns

sns.set()

x = np.linspace(-4, 4, 50)
y = np.sin(x)

plt.plot(x, y)
plt.plot(x, np.cos(x), '*', c='r')

plt.show()
```

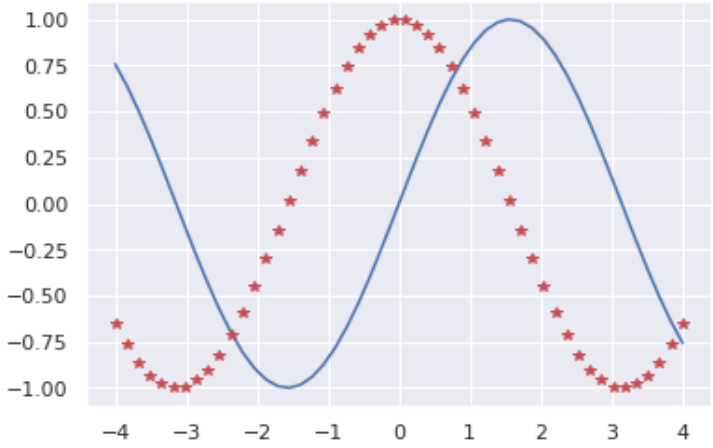


Рис. 1: График функции

Можно отобразить какие-то на плоскости

```
xy = np.random.randint(0, 5, (20, 2))
c = np.random.random_sample(size=20)
plt.scatter(xy[:, 0], xy[:, 1], s=100, c=c)
```

Рис. 2: Точечная диаграмма

Столбиковая диаграмма

```
1  plt.bar([1, 2, 3], [4, 5, 6])
```

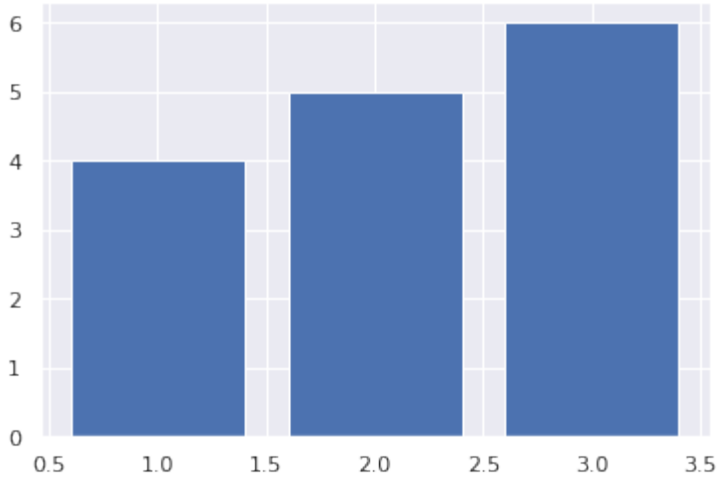


Рис. 3: Диаграмма

И более сложную инфографику.

```
import matplotlib.patches as pc

def func(x):
    return(x - 3) * x

ax = plt.subplot(111)

a, b = 4, 9 # integral area
x = np.arange(0, 10, 0.01)
y = func(x)
plt.plot(x, y, linewidth=1)

ix = np.arange(a, b, 0.01)
iy = func(ix)
verts = [(a,0)] + list(zip(ix,iy)) + [(b,0)]
poly = pc.Polygon(verts, facecolor='0.8', edgecolor='k')
ax.add_patch(poly)

plt.text(a, 40,
     r"$\int_4^9 f(x)\mathrm{d}x$",
         horizontalalignment='center',
     fontsize=20)
```

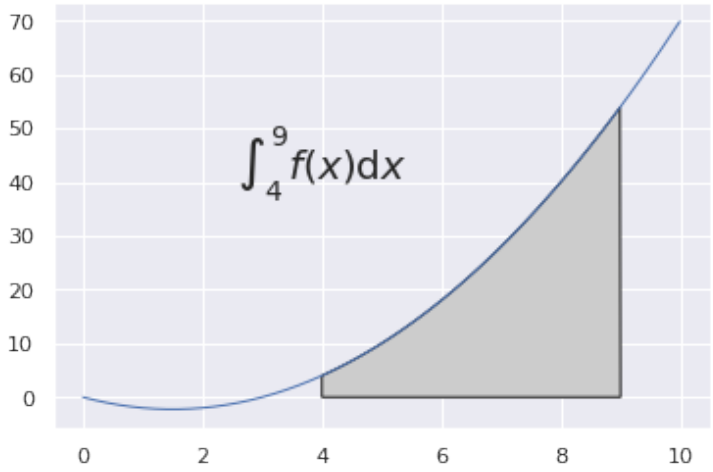


Рис. 4: Визуализация интеграла

SciPy

Пакет `SciPy`[12] - это расширение `NumPy`, которое содержит большое количество функций для инженерных и научных вычислений. Например:

- поиск экстремума функций
- численное интегрирование
- обработка сигналов
- решение систем ОДУ
- семплирование, математическая статистика

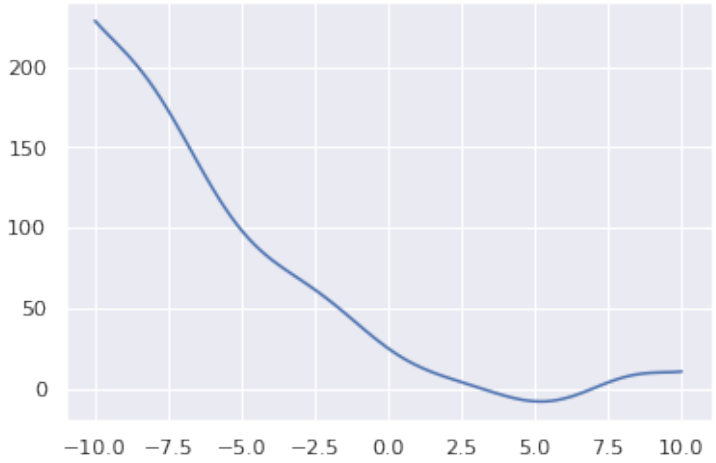
Приведем пример поиска минимум функции:

```
import scipy as sp

def func(x):
    return (x - 2) * np.sin(x) + (x - 5.) ** 2 - x

x = np.linspace(-10, 10, 100)

plt.plot(x, func(x))
```



Применяем реализованный в `SciPy` алгоритм Бройдена — Флетчера — Гольдфарба — Шанно

```
sp.optimize.fmin_bfgs(func, 0)
```

```
Optimization terminated successfully.
         Current function value: -7.987077
         Iterations: 6
         Function evaluations: 27
         Gradient evaluations: 9

array([5.19725835])
```

SymPy

SymPy[13] - очень простая библиотека для символьных вычислений. Выражение задается декларативно с помощью синтаксиса Python. Например, посчитаем производную функции $f(x) = (x-2) \cdot sin(x) + (x-5)^2 - x$

```
1  import sympy
2  from sympy import Symbol, sin, lambdify
3  sympy.init_printing(use_unicode=True)
4
5  sym_x = Symbol('x')
6  sym_y = (sym_x - 2) * sin(sym_x) + (sym_x - 5) ** 2 -
       sym_x
7  sym_diff = sym_y.diff(sym_x)
8
9  sym_y
```

$$-x + (x-5)^2 + (x-2)\sin(x)$$

Воспользуемся тем же самым алгоритмом Бройдена — Флетчера — Гольдфарба — Шанно, но на этот раз зададим производную явным образом

```
1  diff = lambdify(sym_x, sym_diff, 'numpy')
2
3  sp.optimize.fmin_bfgs(func, 0, diff)
```

```
Optimization terminated successfully.
         Current function value: -7.987077
         Iterations: 6
         Function evaluations: 9
         Gradient evaluations: 9

array([5.19725836])
```

Можно посчитать значение производной в какой-либо точке

```
1  diff(5)
```

```
-1.1079377182734595
```

Или построить её график

```
1  plt.plot(x, func(x))
2  plt.plot(x, diff(x))
```

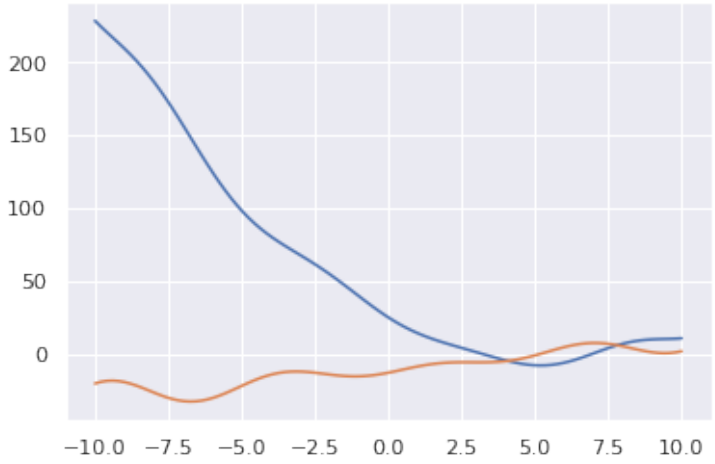


Рис. 5: График функции и производной

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

Задачи

NB. Все упражнения ниже нужно делать без использования циклов Python

Задача 1. Есть 5 точек в декартовой системе координат (в виде матрицы X размерностью 5×2), сконвертируйте эти точки в полярную си-

стему координат.

```
X = np.random.random((5, 2))
```

Задача 2. Найдите индексы максимального элемента в случайной матрице 10×10.

```
X = np.random.random((10, 10))
```

Задача 3. Есть 10 точек (X) и ещё одна (y). Найти в X ближайшую к y точку.

```
X = np.random.random((10, 2))
y = np.random.random((1, 2))
```

Задача 4. Дана функция:

$$\begin{cases} x^2 + 2x + 6, & x < 0 \\ x + 6, & 0 \leq x \leq 2 \\ x^2 + 4x - 4, & x \geq 2 \end{cases}$$

Постройте массив из её значений на $-3 \leq x \leq 3$.

Задача 5. Из каждого элемента матрицы вычесть среднее арифметическое от всех элементов в соответствующей строке (после чего среднее значение каждой строки должно равняться нулю).

```
X = np.random.random((10, 10))
```

Задача 6. Есть массив из 1000 чисел, полученных из генератора случайных чисел, имеющий нормальное распределение. Посчитайте выборочное среднее и выборочную дисперсию.

```
X = np.random.normal(loc=5, scale=2., size=1000)
```

Задача 7. Создайту матрицу:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 0 \\ 2 & 3 & 4 & 0 & 1 \\ 3 & 4 & 0 & 1 & 2 \\ 4 & 0 & 1 & 2 & 3 \end{pmatrix}$$

Pandas

Pandas[5] - мощная библиотека для анализа данных на Python. Позволяет работать с табличными данными, временными рядами и другими структурами данных, анализировать и визуализировать их.

Основные преимущества: - богатый инструментарий для анализа данных - агрегация, трансформация и т.п. - возможность работы (чтения и запись) с различными форматами данных - CSV, Excel, СУБД - совместимость со многими библиотеками из экосистемы Python - возможность работать и иерархическими данными

Series

```
import pandas as pd
```

Базовая структура данных - это `Series`. Интуитивно это типизированный вектор, каждый элемент которого имеет ассоциированную с ним некоторую уникальную метку. Набор меток, который привязан к `Series` называется индексом. `Series` можно создать, например, из списка `Python` или из массива `NumPy`

```
pd.Series([1, 2, 3, 4])
```

```
0    1
1    2
2    3
3    4
dtype: int64
```

по умолчанию в качестве индекса выступают упорядоченные целые числа, но можно использовать строки, даты, и многие другие объекты.

```
s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'
    ])
s
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

Можно и прямо использовать словарь с аналогичным результатом

```
pd.Series({'a': 1, 'b': 2, 'c': 3, 'd': 4})
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

К элементам `Series` можно обращаться по индексу

```
s['c']
```

```
3
```

Так как индекс упорядоченный, мы можем обращаться к элементам в `Series` с помощью чисел, точно так же как мы это делали в списках `Python` или массива `NumPy`

```
s[2]
```

```
3
```

Можем даже использовать срезы, получая новый объект `Series`

```
s[2:4]
```

```
c    3
d    4
dtype: int64
```

Можно обратиться по ключам

```
s['c':'d']
```

```
c    3
d    4
dtype: int64
```

Из объекта `Series` можно получить `NumPy`-массив значений

```
s.values
```

```
array([1, 2, 3, 4])
```

и значения индекса

```
s.index
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

Рассмотрим более интересный пример - загрузим CSV-файл с информацией о курсе Евро с 01.04.2017 ao 15.04.2017

```
# первые 5 строчек нашего файла
!head –n 5 data/eur.csv
```

```
2017–04–01,59.8107
2017–04–04,59.8953
2017–04–05,60.2427
2017–04–06,59.6948
2017–04–07,60.0827
```

Загружаем файл в `Series`

```
s = pd.read_csv('data/eur.csv', index_col=0,
                header=None).loc[:, 1]
s
```

```
0
2017-04-01    59.8107
2017-04-04    59.8953
2017-04-05    60.2427
2017-04-06    59.6948
2017-04-07    60.0827
2017-04-08    60.5687
2017-04-11    60.7469
2017-04-12    60.3042
2017-04-13    60.2631
2017-04-14    60.2867
2017-04-15    59.7791
Name: 1, dtype: float64
```

Элементы индекса - даты

```
1  s.index
```

```
Index(['2017-04-01', '2017-04-04', '2017-04-05',
       '2017-04-06', '2017-04-07', '2017-04-08',
       '2017-04-11', '2017-04-12', '2017-04-13',
       '2017-04-14', '2017-04-15'],
      dtype='object', name=0)
```

Как мы видим, `Pandas` сама определила типы данных (в том числе даты)

Мы можем получить даты, для которых курс Евро меньше 60

```
1  s < 60
```

```
0
2017-04-01     True
2017-04-04     True
2017-04-05    False
2017-04-06     True
2017-04-07    False
2017-04-08    False
2017-04-11    False
2017-04-12    False
2017-04-13    False
2017-04-14    False
2017-04-15     True
Name: 1, dtype: bool
```

Обратите внимание, что результат этой операции - новый `Series`, где индекс остается неизменным, а вместо курса евро - булевское значение. Наличие индекса - это основное отличие `Series` от обычных одномерных массивов `NumPy`.

Следующая стандартная операция заключается в получении подмножество `Series` на основе некоторого условия. Получим подмножество на даты, когда курс меньше 60.

```
1  s[s < 60]
```

```
0
2017-04-01    59.8107
2017-04-04    59.8953
2017-04-06    59.6948
2017-04-15    59.7791
Name: 1, dtype: float64
```

Где курс меньше 60 после 4 апреля 2017 года

```
1  s[(s < 60) & (s.index > '2017-04-04')]
```

```
0
2017-04-06    59.6948
2017-04-15    59.7791
Name: 1, dtype: float64
```

Обратите внимание, что результат двух операций `s < 60` и `s.index > '2017-04-04'` – `Series` с логическими значениями. Мы совершаем между ними логическую операции “&” и получаем ещё один `Series` с логическими значениями.

DataFrame

С помощью `Series` мы можем оперировать одномерными данными (как в примере выше: дата → курс евро). Но если данных больше, и мы хотим производить какой-то анализ на основе курса евро и курса доллара? Мы можем создать два объекта `Series`, но одновременная работа с ними может быть не очень удобной.

DataFrame - это именованный набор объектов `Series` с одинаковым индексом. Если говорить более простым языком - это таблица, у каждой строки которой есть уникальное значение (элемент индекса) и у каждого столбца - имя. Это чем-то напоминает `Excel`, только управлять таблицей мы будем из `Python`. Стоит отметить, что появление `DataFrame` инспирировано одноименной структурой данных из языка `R`.

`DataFrame` можно просто создать, передав в конструктор табличные данные и имена колонок

```
1  pd.DataFrame(data=[[1, 2, 3], [4, 5, 6]], columns=['a
       ', 'b', 'c'])
```

	a	b	c
0	1	2	3
1	4	5	6

Или аналогично по столбцам

```
pd.DataFrame(data={'a' : [1, 4], 'b': [2, 5], 'c':
    [3, 6]})
```

	a	b	c
0	1	2	3
1	4	5	6

Вместо списков, можно использовать массивы `NumPy` или `Series`

Давайте загрузим исторические данные по евро, доллару и британскому фунту. Наш файл (в формате `CSV`) выглядит так

```
!head –n 5 data/currencies.csv
```

```
DATE,EUR,USD,GBP
2017–04–01,59.8107,55.9606,69.7605
2017–04–04,59.8953,56.1396,70.3429
2017–04–05,60.2427,56.5553,70.3548
2017–04–06,59.6948,55.894,69.4986
```

с помощью параметров мы указываем как следует читать файл:

- *index_col=0* - индекс расположен в первой колонке (отсчет идёт, как обычно, с нуля).
- *header=0* - заголовки таблицы на первой строке. Если наш файл не содержит заголовка, мы можем просто указать *header=False*, в этом случае колонки будут носить численные именования.

Чтобы не загромождать экран, мы будем использовать метод *.head()*, который обрезает нашу таблицу до 5 строк.

```
df = pd.read_csv('data/currencies.csv', index_col=0,
    header=0)
df.head()
```

DATE	EUR	USD	GBP
2017-04-01	59.8107	55.9606	69.7605
2017-04-04	59.8953	56.1396	70.3429
2017-04-05	60.2427	56.5553	70.3548
2017-04-06	59.6948	55.894	69.4986
2017-04-07	60.0827	56.4369	70.3655

Основные операции

Мы можем получить один столбец, которые будет представим в виде объекта `Series`

```
df['USD'].head()
```

DATE	USD
2017-04-01	55.9606
2017-04-04	56.1396
2017-04-05	56.5553
2017-04-06	55.894
2017-04-07	56.4369

Или “срезать” `DataFrame`, взяв только некоторые, интересные нам столбцы:

```
df[['USD', 'GBP']].head()
```

DATE	USD	GBP
2017-04-01	55.9606	69.7605
2017-04-04	56.1396	70.3429
2017-04-05	56.5553	70.3548
2017-04-06	55.894	69.4986
2017-04-07	56.4369	70.3655

мы можем создать новый столбец из старых. В данном случае посчитаем разницу между курсом фунта и евро на каждый день

```
df['GBP_EUR'] = df['GBP'] - df['EUR']
df.head(5)
```

DATE	EUR	USD	GBP	GBP-EUR
2017-04-01	59.8107	55.9606	69.7605	9.9498
2017-04-04	59.8953	56.1396	70.3429	10.4476
2017-04-05	60.2427	56.5553	70.3548	10.1121
2017-04-06	59.6948	55.894	69.4986	9.8038
2017-04-07	60.0827	56.4369	70.3655	10.2828

Или можно посмотреть на сколько изменялся курс евро и доллара с прошлого дня

```
df['$\Delta$ EUR'] = df['EUR'] - df['EUR'].shift(1)
df['$\Delta$ USD'] = df['USD'] - df['USD'].shift(1)
df.head()
```

DATE	EUR	USD	GBP	GBP-EUR	Δ EUR	Δ USD
2017-04-01	59.8107	55.9606	69.7605	9.9498	nan	nan
2017-04-04	59.8953	56.1396	70.3429	10.4476	0.0846	0.179
2017-04-05	60.2427	56.5553	70.3548	10.1121	0.3474	0.4157
2017-04-06	59.6948	55.894	69.4986	9.8038	-0.5479	-0.6613
2017-04-07	60.0827	56.4369	70.3655	10.2828	0.3879	0.5429

Индексация осуществляется с помощью свойств **.loc** и **.iloc**. Допустим, мы хотим получить новый `DataFrame`, состоящий из строк, где измене-

ние доллара и евро с прошлого дня отрицательные

```
df.loc[(df['$\Delta$ USD'] < 0)
       & (df['$\Delta$ EUR'] < 0)]
```

DATE	EUR	USD	GBP	GBP-EUR	Δ EUR	Δ USD
2017-04-06	59.6948	55.894	69.4986	9.8038	-0.5479	-0.6613
2017-04-12	60.3042	56.9552	70.7384	10.4342	-0.4427	-0.4344
2017-04-13	60.2631	56.7556	70.9502	10.6871	-0.0411	-0.1996
2017-04-15	59.7791	56.2945	70.4413	10.6622	-0.5076	-0.3074

Если к данным нужно обратиться по номеру строки, то используется свойство **.iloc**

```
df.iloc[0:2]
```

DATE	EUR	USD	GBP	GBP-EUR	Δ EUR	Δ USD
2017-04-01	59.8107	55.9606	69.7605	9.9498	nan	nan
2017-04-04	59.8953	56.1396	70.3429	10.4476	0.0846	0.179

`Pandas` имеет большое количество функций для подсчет статистических характеристик наших данных. Например, посчитаем коэффицент корреляции между значениями доллара и евро

```
df['USD'].corr(df['EUR'])
```

```
0.9529946342029497
```

Или математическое ожидание курсов валют

```
df[['USD', 'EUR', 'GBP']].mean()
```

USD	56.5367
EUR	60.1523
GBP	70.5092

Можно сразу визуализировать наши данные, с помощью специальных функций (используется библиотека `matplotlib`)

Визуализация

```
%matplotlib inline

df[['USD', 'EUR']].plot()
```

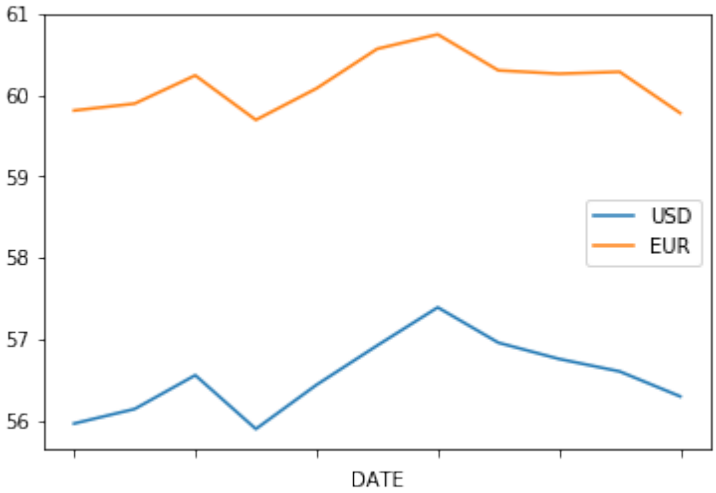


Рис. 6: Курсы валют

Сохранение

`DataFrame` можно сохранить в `CSV`-файл, `Excel` и в другие форматы:

```
1  df.to_csv('data/curr.csv')
2  df.to_excel('data/curr.xsl')
```

Задачи

В файле `weather.csv`[14] находится архив погоды в г. Санкт-Петербург с 1 января 2008 года по 31 декабря 2016 года. Файл состоит из двух столб-

цов - дата (**Day**) и средняя дневная температура в этот день (**t**). Прочитать данные в pandas можно с помощью кода:

```
df = pd.read_csv('weather.csv', encoding='utf-8',
    index_col=False, parse_dates=[0])
```

```
df.head(5)
```

	Day	t
0	2008-01-01	0
1	2008-01-02	-5
2	2008-01-03	-11
3	2008-01-04	-11
4	2008-01-05	-12

С помощью Pandas решите следующие задачи

Задача 1. Определите самый холодный год, в котором среднеяя температура была минимальная и самый теплый год.

Задача 2. Определите год, где в январе было наибольшее число дней с положительной температурой ($t \geq 0$).

Задача 3. В каком году было самое холодное лето (по средней температуре)

Задача 4. Найдите день с самой большим перепадом температуры, если сравнивать со следующим днем.

Задача 5. Постройте график среднегодовых температур (по оси x - год, по оси y - средняя температура в этот год).

Регулярные выражения. Обработка текстов.

Часто возникает задача поиска в тексте каких-то элементов (например гиперссылок) или проверки введенных пользоватем данных. Для упрощения этих задач можно использовать регулярные выражения, реализация которых присутствует в стандартных библиотеках большей части популярных языков программирования, в том числе и в `Python`.

Наиболее типичные сценария использования:

- поиск паттерна в строке
- проверка строки на совпадение паттерну
- сегментация строки по паттерну
- замена паттерна в строке.

Паттерн описывается с помощью специального языка - регулярного выражения, в `Python` используются регулярные выражения со следующими базовыми наборами правил:

.	любой символ	\d	цифра
\D	не цифра	\s	пробельный символ
\S	не пробельный символ	\w	буквенный символ
\W	не буквенный символ	^	начало строки
$	конец строки	\b	начало слова
\B	конец слова	[abc]	символ из перечисленных

[^abc]	кроме символов	[a-zA-Z]	символы из интервалов
X|Y	или		

Наборы символов и символьных классов можно объединять в группы, наподобие того как это происходит в обычных математических выражениях. Для описания групп с различными целями существует следующие конструкции

(X)	группа (capturing)	(?:X)	группа (non-capturing)
(?=X)	предпросмотр	(?!X)	негативный предпросмотр

Квантификатор после какого-то подвыражения (символа или группы) позволяет задать правила повторения этого подвыражения. Квантификатор может относиться более чем к одному символу в регулярном выражении только если это группа. В Python определены жадные квантификаторы

X?	0 или 1 повторение	X*	≥ 0 повторений
X+	≥ 1 повторений	X{n}	ровно n повторений
X{n,}	$\geq n$ повторений	X{n,m}	от n до m повторений

и ленивые квантификаторы

X??	0 или 1 повторение	X*?	≥ 0 повторений

X+?	≥ 1 повторений	X{n}?	ровно n повторений
X{n,}?	$\geq n$ повторений	X{n,m}?	от n до m повторений

Разница заключается в алгоритме сопоставления - жадные квантификаторы пытаются сопоставить как можно больше символов входной строки, ленивые - как можно меньше.

Реализации регулярных выражений

Коснемся темы реализации движков для работы с регулярными выражениями.

Формальный язык — множество конечных слов над конечным алфавитом Σ. Пусть есть некоторое конечно множество символов Σ, тогда множество $L \in \Sigma^*$ есть формальный язык.

Над формальными языками можно определить операции:

- $L_1 \cap L_2$
- $L_1 \cup L_2$
- $L_1 \quad L_2$
- $L_1 ⊠L_2 $ - новый язык, в котором ко всем возможным словам из L_1 присоеденены справа слова из L_2
- L^* - замыкание клини, $\{\epsilon\} \cup L \cup (L \cdot L) \cup (L \cdot L \cdot L) \cup \cdots$

Формальный язык над алфавитом Σ является **регулярным**, если он принадлежит множеству языков $R \in \Sigma^*$:

- $\emptyset \in R$

- $\{\varepsilon\} \in R$
- $\forall a \in \Sigma : \{a\} \in R$
- $P \in R \wedge Q \in R \Rightarrow (P \cup Q) \in R$
- $P \in R \wedge Q \in R \Rightarrow (P \cdot Q) \in R$
- $P \in R \Rightarrow P^* \in R$

Любой регулярный язык может быть описан:

- детерменированным конечным автоматом
- недетерменированным конечным автоматом
- регулярным выражением
- регулярном грамматикой

Конечные автоматы

Конечный автомат это упорядоченная пятерка $A = (\Sigma, Q, q_0, F, \delta)$, где

- Σ - входной алфавит
- Q - множество состояний
- $q_0 \in Q$ - начальное состояние
- $F \subset Q$ - множество конечных состояний
- $\delta : (\Sigma \cup \varepsilon) \times Q \to 2^Q$ - функция перехода

В зависимости от определения функции перехода:

- недетерминированный конечный автомат с ε-переходами (ε-NFA)
$$\delta : (\Sigma \cup \varepsilon) \times Q \to 2^Q$$
- недетерминированный конечный автомат (NFA)
$$\delta : \Sigma \times Q \to 2^Q$$

- детерминированный конечный автомат (DFA)

$$\delta : \Sigma \times Q \to Q$$

Множество слов, которые принимаются конечным автоматом образуют регулярный язык. По любому ε-NFA можно построить эквивалентный DFA. В DFA можно минимизировать число состояний.

Графическое отображение конечного автомата представлено на изображении

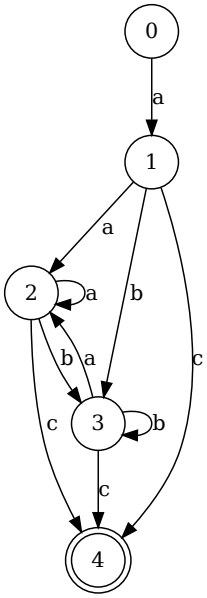


Рис. 7: Конечный автомат

Существует два распространенных способа реализации регулярных выражений, применяемых в различных задачах (не считая гибридов и т.п.):

1. регулярное выражение -> ε-NFA -> DFA -> min-DFA
2. backtracking

В стандартных библиотеках популярных языков программирования применяется backracking.

Разновидности синтаксиса регулярных выражений

Исторически существует несколько возможных синтаксисов языков для описания регулярных выражений

- POSIX RE[9] (`.`, `*`, `[ ]`, `[^ ]`, `\{ \}`, `\( \)`)
- POSIX ERE (`.`, `*`, `+`, `?`, `|`, `[ ]`, `[^ ]`, `{ }`, `( )`)
- PCRE[6] (стандартные библиотеки `Perl`, `Java`, `C#`, `Python`)

Реализация в языке Python

В `Python` стандартные регулярные выражения определены в модуле `re`[11]. Кроме того, существует специальный синтаксис задания строк для регулярных выражений - префикс `r`. В строках с этим префиксом не требуется специальным образом задавать специальные управляющие символы (например, `,`)

```
1  import re
```

Рекомендует сначала скомпилировать регулярное выражение с помощью функции `re.compile()`, в результате получится специальный

объект, который можно использовать для выполнения основных задач

```
pattern = re.compile(r'a+a$')
```

Например, чтобы проверить совпадает ли паттерн, заданный регулярным выражением, со строкой, можно использовать метод `fullmatch`

```
m = pattern.fullmatch('aaaa')
```

В данном случае строка `'aaaa'` соответствует паттерну `'a+a$'`, возвращается объект специального типа. Если соответствия нет, то возвращается `None`.

Второй возможный сценарий использования - поиск непересекающихся подстрок, которые соответствую паттерну. Введем паттерн - все слова из букв *a* которые заканчиваются на *b*.

```
pattern = re.compile(r'a+b')
```

Найдем все вхождения этого паттерна в строке:

```
s = 'Hello aaab world ab !'
pattern.findall(s)
```

```
['aaab', 'ab']
```

Ещё один сценарий - разделение строки на подстроки по паттерну. Это бывает полезным, когда нужно быстро разделить, например, предложения на слова, убрав все знаки препинания.

```
pattern.split(s)
```

```
['Hello ', ' world ', ' !']
```

Следующий сценарий использования - замена вхождений паттерна в строке.

```
pattern.sub('e', s)
```

```
'Hello e world e !'
```

В качестве замены можно передать не только строку, но и `lambda` -функцию, которая будет вызываться на каждое совпадение с образцом. В данном случае мы добавляем к каждому совпадению букву *a*.

```
pattern.sub(lambda x: x.group(0) + 'a', s)
```

```
'Hello aaaba world aba !'
```

Приведем более сложный пример. В качестве паттерна опишем адрес электронной почты (упрощенный, данный паттерн не удовлетворяет спецификации).Дальше разобьем этот адрес на составляющие:

```
pattern = re.compile(r'(\w+)@(\w+)\.(\w{2,3})')

matcher = pattern.match('test@example.com')
if matcher:
    print(matcher.group(0))
    print(matcher.group(1))
    print(matcher.group(2))
    print(matcher.group(3))
```

```
test@example.com
test
example
com
```

Можно найти все вхождения электронной почты в тексте и напечатать только домены первого уровня:

```
t = 'test@example.com ssss test2@gmail.com'

pattern = re.compile(r'(\w+)@((\w+)\.(\w{2,3}))')
for m in pattern.finditer(t):
    print (m.group(2))
```

```
example.com
gmail.com
```

или имена пользователей:

```
s = 'test@example.com hello@mail.ru'
matchers = pattern.finditer(s)
for matcher in matchers:
    print (matcher.group(1))
```

```
test
hello
```

Обратим внимание на работу жадных и ленивых квантификаторов. В случае применения жадных результат может быть не тем, что ожидается

```
s = '<h1> text1 </h1>  <h2> text3 </h2>'
re.findall(r'<\w+>(?:.+)</\w+>', s)
```

```
['<h1> text1 </h1>  <h2> text3 </h2>']
```

В данном случае можно использовать ленивые

```
1  re.findall(r'<\w+>(?:.+?)</\w+>', s)
```

```
['<h1> text1 </h1>', '<h2> text3 </h2>']
```

Ещё один важный момент касается производительности движка регулярных выражений. Так как в Python регулярные выражения реализованы через механизм backtracking'а, то в некоторых случаях можно получить экспоненциальную сложность.

```
1  re.findall(r'(a*a*)*c', 'a' * 10 + 'e')
```

В модуле re есть недокументированный класс Scanner, с помощью которого можно реализовать лексический анализатор. Scanner будет искать вхождения паттернов в тексте и на каждое совпадение вызывать соответствующую функцию. В общем случае подобный код неэффективен, лексические анализаторы лучше реализовывать с помощью специальных инструментов - генераторов лексических анализаторов, которые обеспечат анализ за линейное время.

```
1  scanner = re.Scanner(
2     [(r'(\w+)@(\w+)\.(\w{2,3})', lambda s, x: (x, '
          email')),
3      (r'[a-zA-Z]+', lambda s, x: (x, 'word')),
4      (r'\d+', lambda s, x: (x, 'digit')),
5      (r'\s+', lambda s, x: (x, 'whitespace')),
6      (r'[.,;"!?:]', lambda s, x: (x, 'preposition')),
7      ])
8
9  scanner.scan('hello, world 1234 test@example.com')
```

```
([('hello', 'word'),
  (',', 'preposition'),
  (' ', 'whitespace'),
  ('world', 'word'),
  (' ', 'whitespace'),
  ('1234', 'digit'),
  (' ', 'whitespace'),
  ('test@example.com', 'email')],
 '')
```

Инструменты для обработки текстов

Привидем список некоторых полезных библиотек для обработки текстовых данных:

- ply[8] (Python) - библиотека для написания лексических анализаторов на Python
- pyparsing[10] (Python) - библиотека для написания синтаксических анализаторов на Python
- lex, flex (C) - классические библиотеки - генераторы лексических анализаторов
- jflex[2] (Java) - библиотека для написания лексических анализаторов на Java
- ANTLR[1] (Java, C++, Python)

Приведем лексического анализатора на ply[8]. В данном случае анализатор описывается в классе, могут быть три вида токенов - слова, цифры и пробелы.

```
from ply.lex import lex, TOKEN

class Lexer:
    tokens = ( 'NUMBER', 'ID', 'WHITESPACE' )

    @TOKEN(r'\d{1,5}')
    def t_NUMBER(self, t):
        t.value = int(t.value)
        return t

    @TOKEN(r'\w+')
    def t_ID(self, t):
        return t

    @TOKEN(r'\s+')
    def t_WHITESPACE(self, t):
        pass

    def t_error(self, t):
        pass


__file__ = ''      # make `ply` happy

lexer = lex(object=Lexer())
lexer.input('123 abs 965')
for token in lexer:
    print(token)
```

Для каждого токена в тексте выозвращается необходимая информация - типа, длина смещение:

```
LexToken(NUMBER,123,1,0)
LexToken(ID,'abs',1,4)
LexToken(NUMBER,965,1,8)
```

Другой пример `pyparsing`[10], с помощью которого можно обрабатывать более широкий класс формальных языков. С помощью специального DSL (domain-specific language, предметно-ориентированный

язык) описывается грамматика. С помощью `pyparsing` можно обрабатывать коллекции в специфичных форматах, извлекать логи и так далее:

В данном примере грамматика описывает строку, которая начинается со слова, после которого идет двоеточие и набор чисел через запятую.

```
from pyparsing import Word, alphas, nums,  Literal,
    StringEnd, ZeroOrMore, Suppress, OneOrMore

word = Word(alphas)
num = Word(nums)
sep = Suppress(OneOrMore(','))
col = Suppress(':')

s = word + col + num + ZeroOrMore(sep + num) +
    StringEnd()

s.parseString('hello: 1, 22, 3')
```

```
(['hello', '1', '22', '3'], {})
```

Здесь более сложный пример, грамматика описывает правильные скобочные записи.

```
from pyparsing import Literal, Forward, StringEnd,
    OneOrMore, Empty

br_o = Literal('(')
br_c = Literal(')')

braces = Forward()
braces << OneOrMore(br_o + (braces | Empty() ) + br_c
    )
start = braces + StringEnd()

start.parseString('(())()()')
```

```
(['(', '(', ')', ')', '(', ')', '(', ')'], {})
```

И, наконец, простейшие математические выражения с приоритетом операций. Сначала можно определить классы, которые будут узлами дерева выражения

```
from pyparsing import Word, Literal, Or, nums,
    Forward, StringEnd
from operator import mul, truediv, add, sub

class NumNode(object):
    def __init__(self, t):
        self.num = float(t[0])
    def calc(self):
        return self.num
    def __repr__(self):
        return 'Num(%s)' % self.num

class OpNode(object):
    def __init__(self, t):
        self.left = t[0]
        self.op = { '-' : sub, '+' : add, '/' :
            truediv, '*' : mul }[t[1]]
        self.right = t[2]
    def calc(self):
        return self.op(self.left.calc(), self.right.
            calc())
    def __repr__(self):
        return 'Op(%s, %s, %s)' % (self.left, self.op
            , self.right)
```

Затем грамматику

```
plus = Literal('+')
minus = Literal('-')
div = Literal('/')
mult = Literal('*')

factor = Word(nums).setParseAction(NumNode)

term = Forward()
term << (( factor + (mult | div) + term ).
    setParseAction(OpNode) | factor )

expr = Forward()
expr << ((term + (plus | minus) + expr).
    setParseAction(OpNode) | term )

start = expr + StringEnd()
```

Применение

```
tree = start.parseString('2 * 4 + 6 * 7')[0]
print(tree)
print(tree.calc())
```

```
Op(Op(Num(2.0), <built-in function mul>, Num(4.0)), <
    built-in function add>, Op(Num(6.0), <built-in
    function mul>, Num(7.0)))
50.0
```

NLTK

Коснёмся одной библиотеки - Natural Language Toolkit[4], библиотека для обработки естественных языков. Она создавалась для учебных целей, но тем не менее приобрела определенную популярность. Например в ней реализовано множество методов токенизации, которые можно использовать для повседневных задач и экспериментов.

```
import nltk
from nltk.tokenize import wordpunct_tokenize,
    word_tokenize

nltk.download('punkt')

(word_tokenize('Hello world 4.2.'),
 word_tokenize('LA New—York'),
 wordpunct_tokenize('Hello world 4.2!'))
```

```
(['Hello', 'world', '4.2', '.'],
 ['LA', 'New—York'],
 ['Hello', 'world', '4', '.', '2', '!'])
```

Задачи

Задача 1. Реализуйте регулярное выражение, которое описывает упрощенное url.

Задача 2. Реализуйте функцию, которая заменяет все вхождения электронных адресов в тексте на что-то обезличенное, например *user@example.com* на *xxxx@example.com*

Список литературы

1. ANTLR [Электронный ресурс] / (01.03.2020). Режим доступа: https://antlr.org.

2. JFlex [Электронный ресурс] / (01.03.2020). Режим доступа: http://jflex.de.

3. matplotlib [Электронный ресурс] / (01.03.2020). Режим доступа: https://pypi.python.org/pypi/matplotlib/.

4. NLTK [Электронный ресурс] / (01.03.2020). Режим доступа: https://nltk.org.

5. Pandas [Электронный ресурс] / (01.03.2020). Режим доступа: https://pypi.python.org/pypi/pandas.

6. PCRE [Электронный ресурс] / (01.03.2020). Режим доступа: https://pcre.org/pcre.txt.

7. Plotly [Электронный ресурс] / (01.03.2020). Режим доступа: https://plot.ly.

8. Ply [Электронный ресурс] / (01.03.2020). Режим доступа: https://github.com/dabeaz/ply.

9. POSIX RE [Электронный ресурс] / (01.03.2020). Режим доступа: https://www.regular-expressions.info/posix.html.

10. pyparsing [Электронный ресурс] / (01.03.2020). Режим доступа: https://github.com/pyparsing/pyparsing.

11. Regular expression operations [Электронный ресурс] / (01.03.2020). Режим доступа: https://docs.python.org/3/library/re.html.

12. SciPy [Электронный ресурс] / (01.03.2020). Режим доступа: https://scipy.org/.

13. SymPy [Электронный ресурс] / (01.03.2020). Режим доступа: http://www.sympy.org/en/index.html.

14. Статистика погоды [Электронный ресурс] / (01.03.2020). Режим доступа: https://bitbucket.org/snippets/alms/KrG4LL.